状态逻辑型程序设计语言*

阎志欣

(北京航空航天大学计算机科学与工程系,北京 100083)

摘要 本文提出了一种新的纯逻辑式子句型程序设计语言.文中给出了语言的语法,非形式语义,子句的过程解释和基于约束归结的推理系统.对该语言来说,程序包含三类变量:输入变量,输出变量和用于控制机器资源的程序变量;被程序定义的函数符号可用于构造项或子项,并且还可用作为谓词符号;不需要低效的最广合一.由于这些因素,一个子句集本身隐含了顺序,分支,迭代和递归多种控制结构使得容易构造高效的定理证明系统.这种语言将是一种有坚实理论基础的,高效的,实际有用的高级确定性语言.

关键词 状态逻辑语言,调用规则,分裂函数调用,约束归结,确定性.

程序设计语言的目的是对待求解问题的算法实现人—机通讯.算法由逻辑和控制两个成分组成.逻辑成分是求解问题时所用信息之间的逻辑关系.控制成分是求解问题时使用信息的方式.用传统的指令式语言,程序把算法的逻辑成分和控制成分混合交错,使逻辑被淹没在控制中,程序失去了有用的数学特性,从而导致了不易理解、分析和证明等许多问题.1978年Backus在他的图灵奖演讲中对该类语言存在的严重问题有详尽而全面的分析[1].

近20年来把算法的逻辑成分从控制成分中分离出来的陈述式语言被广泛研究.Darlington, Guo, Lock和Pull在文献[2—4]中对各种陈述式语言的语法,语义,操作模型等有详细讨论.自从Kowalski[5]于1974年提出谓词逻辑可作为程序设计语言的预见以后,基于Horn子句的各种逻辑型语言的研究成为研究陈述式语言的重要分支.纯Horn子句集可以分离算法的逻辑和控制,程序仅描述算法的逻辑,把控制留给执行系统.控制,如顺序、分支、迭代、递归和存贮空间的使用方式隐含使用时空资源的效率.然而基于Horn子句的纯逻辑型语言,由于程序中的变量仅包含输入和输出变量,不包含任何程序变量,这使得程序仅是描述输入输出关系的逻辑.程序员没有足够的信息以控制程序的执行顺序,也没有中间变量可用来存贮中间结果.程序的基本控制结构是递归,难以对迭代给出纯逻辑的描述.这种情况使程序执行系统难以挤压出效率.对该类语言的效率问题,Kowalski在文献[5]中已有分析.对纯函数型陈述式语言也存在同样问题[6].

20年来陈述式语言的效率问题一直是研究的重要问题.有3条途径:第1,并行计算提高执行速度,这仅仅用空间资源换取了速度,并没有解决效率问题本身.第2,在程序中引入

* 本文1993—03—16定稿

本课题得到国家863高技术项目基金资助.作者阎志欣,58岁,教授,主要研究领域为计算、推理理论和语言.

本文通讯联系人:阎志欣,北京 100083,北京航空航天大学计算机科学与工程系

强制性控制指令,如 Prolog 中的 cut 和 fail 及 Lisp 中的 Prog 函数.这一途径可控制程序的执行顺序和进行迭代计算,然而破坏了程序的数学特性.显然,这与陈述式语言的宗旨,即程序员不需要关心程序是如何执行的,相违背.第 3,尾递归优化,存在着可迭代计算但非尾递归的函数不能优化.

本文提出的状态逻辑是一种纯逻辑型语言.其程序不仅包含输入和输出变量,而且包含用于控制机器资源的程序变量,因而程序是描述输入,输出和控制的逻辑;由程序定义的函数符号不仅用于构造项或子项而且可作为谓词符号构造原子谓词.这些因素使得不包含任何强制性指令的状态逻辑本身隐含了顺序,分支,迭代和递归多种控制结构,再加上替换 θ 仅是由一个调用向一个过程单向地传递参数,不需要低效的最广合一算法,因而很容易构造高效的执行系统.我们相信该语言是一种有坚实理论基础,高效的,实际有用的,高级的确定性程序设计语言.

状态逻辑是一种逻辑型语言,然而程序描述的是函数,任何图灵机可计算的对象都通过函数进行.本文仅考虑状态逻辑作为程序设计语言的主要方面.形式语义,编译原理和可计算性有待进一步讨论.

1　语法和语义

一阶逻辑中与逻辑蕴含有关的问题可以通过演示其子句形语句的不可满足性解决[7].子句形语句语法简单而又不失谓词演算的表达能力.状态逻辑语言是一种子句形语言.

定义.项被归纳地定义如下

(1)一个变量是一个项.

(2)一个常量是一个项.

(3)如果 f 是一个 n 元初始或定义函数符号并且 $t_1,\cdots,t_n$ 是项,则 $f(t_1,\cdots,t_n)$ 是一个项.

定义.让 $t_1,\cdots,t_n$ 是项,如果 p 是 n 元定义谓词符号,则 $p(t_1,\cdots,t_n)$ 是一个原子谓词,简称原子.如果 p 是 n 元初始关系符号,则 $p(t_1,\cdots,t_n)$ 是一个约束谓词,简称约束原子.

不含变量的项或原子被称为基础项或基础原子.

定义.形式如下的子句是断言子句

(1) $\rightarrow A$

是事实子句,被看作一个事实的断言.其中 A 是基础原子.

(2) $A,C_1,\cdots,C_n\rightarrow B \quad n\geqslant 0$

是规则子句,被看作一个规则断言.让 $Var(A)$,$Var(C_i)$ 和 $Var(B)$ 分别是 A,C_i 和 B 中的变量集,则子句应满足下面条件:

① B 是原子,A 是仅以变量作为项的原子.

② C_i 是由约束原子和符号"$\rightarrow$"构成的约束文字,"$\rightarrow$"表示"非".

③ $Var(A)=Var(C_1)\cup\cdots\cup Var(C_n)\cup Var(B)$

子句的意义是"对子句中的所有变量,如果 A 是真并且 $C_1,\cdots,C_n$ 都是真,则 B 是真".

(3) $E\rightarrow$

是目标断言.其意义是"对所有变量,E 不是真".其中 E 是仅以变量作为项的原子.

(4) □

是空子句.它被解释为矛盾.

这里(2)—(3)是程序子句.(1)和(4)是由计算过程产生的子句.在状态逻辑中,变量集,定义谓词符号集和定义函数符号集是三个相交的集合.即对同一个定义符号 f,如果 f 用于构造形如 $f(t_1,\cdots,t_n)$ 的项(或子项),则它被解释为函数符号;如果 f 用于构造形如 $f(t_1,\cdots,t_n)$ 原子,则它被解释为谓词符号;如果 f 自身用作项,则它被解释为变量.

定义. 一个状态逻辑程序是程序子句的一个有限集合.

2 例:迭代计算阶乘函数

让原子谓词 $fact(x)$ 表示 $x\geqslant 0$,断言阶乘函数的定义域;原子谓词 $factp(x,i,y)$ 表示 $x\geqslant 0\wedge i\leqslant x\wedge i!=y$,断言程序的中间状态;原子谓词 $factout(x,y)$ 代表 $x\geqslant 0\wedge x!=y$,断言阶乘函数的输入输出关系.

(F1) $fact(x)\rightarrow factp(x,0,s(0))$

(F2) $factp(x,i,y),(i<x)\rightarrow factp(x,s(i),times(s(i),y))$

(F3) $factp(x,i,y),\neg(i<x)\rightarrow factout(x,y)$

(F4) $factout(x,fact)\rightarrow$

把 $s(0),s(1),\cdots$ 看作 $1,2,\cdots$. $times$ 是"乘"运算的函数符号.根据各原子谓词的上述意义不难验证(F1)—(F3)中每个子句的正确性.它们分情况地描述了程序的前置状态和后置状态的逻辑关系.(F4)的意义是对任何 x 和 $fact$,$x\geqslant 0\wedge x!=fact$ 不成立.给定一个计算函数 $times$ 的子句集,则(F1)—(F4)构成了一个计算阶乘的迭代程序.其中 x 是输入变量,i 和 y 是程序变量,函数名 $fact$ 是输出变量.为了计算 2 的阶乘,我们加一个子句

(F0) $\rightarrow fact(2)$

它断言 $2\geqslant 0$ 这一事实.(F0)—(F3)逻辑上蕴含 $fact(2)$ 的值是 2,这与(F4)矛盾.证明过程从事实(F0)出发找这个逻辑矛盾,当它被找到时则产生空子句□,并且以输出变量 $fact$ 返回计算结果 2.

3 例:递归计算 ackermann's 函数

ackermann's 函数的递归方程组是

(1) $ack(0,y)=s(y)$

(2) $ack(x,0)=ack(pred(x),s(0))$

(3) $ack(x,y)=ack(pred(x),ack(x,pred(y)))$

让原子谓词 $ack(x,y)$ 表示 $x\geqslant 0\wedge y\geqslant 0$,断言该函数的定义域;原子谓词 $ackout(x,y,z)$ 表示 $x\geqslant 0\wedge y\geqslant 0\wedge ack(x,y)=z$,断言该函数的输入输出关系.则程序为

(F1) $ack(x,y),(x=0)\rightarrow ackout(x,y,s(y))$

(F2) $ack(x,y),\neg(x=0),(y=0)\rightarrow ackout(x,y,ack(pred(x),s(0)))$

(F3) $ack(x,y),\neg(x=0),\neg(y=0)\rightarrow ackout(x,y,ack(pred(x),ack(x,pred(y))))$

(F4) $ackout(x,y,ack)\rightarrow$

把 $pred(1),pred(2),\cdots$ 看作 0,1,…. 根据 ackermann's 函数递归方程组和各原子谓词的意义不难验证(F1)－(F3)中每个子句的正确性. 这 3 个子句分情况地描述了程序的前置状态和后置状态的逻辑关系. (F4)的意义是对任何 x,y 和 $ack,x\geqslant 0\wedge y\geqslant 0\wedge ack(x,y)=ack$不成立. (F1)－(F4)构造了计算该函数的递归程序. 其中 x,y 是输入变量,函数名 ack 是输出变量. 递归程序不包含任何程序变量. 为了计算当 $x=1,y=1$ 时 ackermann's 函数值,我们加一个子句

(F0) $\rightarrow ack(1,1)$

它断言了 $1\geqslant 0\wedge 1\geqslant 0$ 这一事实. (F0)－(F3)逻辑上蕴含 $ack(1,1)$ 的值是 3,这与(F4)矛盾. 证明过程从事实(F0)出发找这个逻辑矛盾,当找到时则产生空子句□,并且以输出变量 ack 返回计算结果 3.

ackermann's 函数是一个非原始递归,但完全可计算的双递归函数. 它的时空增长速度比原始递归高的多. 提高多重递归效率的有效途径是用迭代减少递归重数. 这要求程序设计语言具有同时表达递归和迭代的能力. 下面是单递归＋迭代的程序.

(F1′) $ack(x,y),(x=0)\rightarrow ackp(x,y,y,s(y))$

(F2′) $ack(x,y),\neg(x=0)\rightarrow ackp(x,y,0,ack(pred(x),s(0)))$

(F3′) $ackp(x,y,i,z),(i<y)\rightarrow ackp(x,y,s(i),ack(pred(x),z))$

(F4′) $ackp(x,y,i,z),\neg(i<y)\rightarrow ackout(x,y,z)$

(F5′) $ackout(x,y,ack)\rightarrow$

这里原子谓词 $ackp(x,y,i,z)$表示 $x\geqslant 0\wedge y\geqslant 0\wedge i\leqslant y\wedge ack(x,i)=z$,断言程序的中间状态. 其它原子谓词的意义如前面所述.

4　过程解释

给状态逻辑程序以过程解释,对书写程序或将谓词逻辑的一般子句 $A_1,\cdots,A_n\rightarrow B_1,\cdots,B_m$ 变换成状态逻辑的子句是有用的. 状态逻辑的子句集被解释为函数声明. 子句集中的子句被过程地解释如下

(1)$A_1,C_1,\cdots,C_n\rightarrow B$ 被解释为过程声明,简称过程. A 被解释为过程名;B 被解释为过程体. 过程体是一个包含函数调用的过程调用. C_i 被解释为包含函数调用的测试条件. 当多个同名过程匹配一个调用时,测试条件为真的过程的过程体被执行. 一个过程体或测试条件中由定义函数符号构成的项或子项被解释为一个函数调用.

(2)$\rightarrow A$ 被解释为一个函数调用,又称起始语句或起始断言.

(3)$E\rightarrow$被解释为一个输出语句. 一个输出语句可以被看作一个过程体为空的特殊过程. 当它被调用时,则以函数名作为变量返回一个函数调用的计算结果.

(4)空子句□被解释为停机语句,被看作一个被满足的目标语句,即带有空体的,没有过程名和测试条件的语句.

5　带函数调用的约束归结

演示子句形语句不可满足性使用基于归结的推理系统是有用的. 用过程解释,归结被解

释为过程调用.归结用于状态逻辑子句时需要考虑约束条件和函数调用规则.

定义. 一个函数调用规则是一个从项到子项的映射函数.函数的值是那个项中的被选子项并且每个被选子项被称为一个函数调用.

下面是几个函数调用规则

LI:最左最内规则(按值调用);L:最左规则(按名调用);PO:并行最外规则;PI:并行最内规则;FA:无宗量规则;FS:全替换规则.

我们以箭头"↑"标示被选的函数调用,add 表示初始函数"加"运算,F 表示任意的定义函数符号,则可用项

add(F(0,F(1)),F(F(2),F(3)))

为例显示上面的函数调用规则

LI: add(F(0,F(1)),F(F(2),F(3)))
↑

L: add(F(0,F(1)),F(F(2),F(3)))
↑

PO: add(F(0,F(1)),F(F(2),F(3)))
↑ ↑

PI: add(F(0,F(1)),F(F(2),F(3)))
↑ ↑ ↑

FA: add(F(0,F(1)),F(F(2),F(3)))
↑ ↑ ↑ ↑

FS: add(F(0,F(1)),F(F(2),F(3)))
↑ ↑ ↑ ↑ ↑

上述调用规则用于传统指令式语言计算递归函数在文献[8]中有详细讨论.显然上述调用规则中 LI 和 L 每步计算仅指定一个子项.所有子项之间的计算顺序由 LI 或 L 唯一地确定.其他调用规则可以被分解为多个可独立执行的基于 LI 和 L 的计算.函数调用规则就是从过程体或约束原子中分离函数调用的规则.状态逻辑的推理规则用于从旧断言序列推导新断言序列.仅由于篇幅限制,本文只给出在 LI 下的推理规则.

定义. 让 C 是一个程序子句,A 是一个基础原子.一个替换 θ 是一个形如 $\{v_1/t_1,\cdots,v_n/t_n\}$ 的有限集合,其中每个 v_i 是 C 中一个的变量,每个 t_i 是 A 中的一个基础项并且变量 $v_1,\cdots,v_n$ 是不同的.

让 S 是一个状态逻辑程序,C 是一个基础约束文字序列,E 是一个断言序列,则我们有下面的推理规则

(1) $C-R_1$ 规则——约束归结规则 1,用于选过程

让 A' 是一个调用,如果存在一个子句 $A,C_1,\cdots,C_n\rightarrow B\in S$ 和替换 θ 使得 $\theta A=A'$ 则有

$$\frac{\rightarrow A';E}{\rightarrow\theta C_1,\cdots,\theta C_n\rightarrow\theta B(A');E}$$

(2) $C-R_2$ 规则——约束归结规则 2,用于选过程体

让 R 是不含函数调用的基础约束文字,A' 是一个调用,则有

$\dfrac{\rightarrow R,C\rightarrow B(A');E}{\rightarrow C\rightarrow B(A');E}$;当 R 为真,C 不空　　$\dfrac{\rightarrow R,C\rightarrow B(A');E}{\rightarrow B;E}$;当 R 为真,C 空

$\dfrac{\rightarrow R,C\rightarrow B(A');E}{\rightarrow A';E}$;当 R 为假

(3) $S-F$ 规则——分离函数调用规则

让 $A(\cdots,h(\cdots t\cdots),\cdots)$是一个断言序列中第一个基础原子或约束文字,$h$ 是初始函数符号,t 是在 LI 下被选函数调用并且它的函数名是 f,则有

$$\frac{\rightarrow A(\cdots,h(\cdots t\cdots),\cdots)}{\rightarrow t;A(\cdots,h(\cdots f\cdots),\cdots)}$$

(4) R 规则——输出项返回规则

让基础原子 $A'(\cdots,a,\cdots)$是一个断言序列中不包含函数调用的第一个原子并且 $B(\cdots f\cdots)$是其后的一个原子或约束文字.如果存在一个形如 $A(\cdots,f,\cdots)\rightarrow\in S$ 的子句和替换 θ 使得 $\theta A(\cdots,f,\cdots)=A'(\cdots,a,\cdots)$并且有替换 $f:=a\in\theta$,则有

$$\frac{\rightarrow A'(\cdots,a,\cdots);B(\cdots f\cdots)}{\rightarrow B(\cdots a\cdots)}$$

由替换 θ 的定义可知,状态逻辑在推理过程中替换仅是一个基础原子向一个程序子句的单向参数传递,因而产生替换 θ 的算法变得非常简单有效.这与传统归结原理中的替换不同,不需要复杂低效的最广合一[9].

6　计　算

对状态逻辑程序计算的概念是,以停机语句为最终对象,重复地使用函数调用,过程调用和约束判定,由旧的断言序列推出新的断言序列.更精确的我们有下面定义.

定义. 让 S 是一个状态逻辑程序,A_1 是起始断言,并且 R 是一个函数调用规则.$S\cup\{A_1\}$的一个经过 R 的计算是一个由断言或断言序列组成的序列 $A_1,\cdots,A_n$ 使得 A_{i+1}是由断言或断言序列 A_i 在 R 下通过推理规则产生的新断言或断言序列.

定义. $S\cup\{A_1\}$的一个经过 R 的反驳是 $S\cup\{A_1\}$的经过 R 的一个有限计算并且该计算中的最后断言序列 A_n 是空子句□.

一个计算可能是有限的或无限的.一个有限的计算可能是成功的或失败的.一个成功的计算恰是一个反驳.一个失败的计算其最后的断言序列 A_n 由于在 S 中没有过程名与其第一个断言匹配而使计算终止.失败的计算意味程序是不完全的.下面我们仅显示在 LI 下的反驳过程.

图 1 显示了由第 3 节中的递归程序(F1)—(F4)和起始断言$\rightarrow ack(1,1)$确定的成功计算.在每个断言序列中的第一个断言是首先要计算的被选断言,以符号"→"指示.连接断言序列 A_i 和 A_{i+1}的弧被标以由 A_i 推出 A_{i+1}的过程或标以函数分离规则.项对函数名的赋值是替换 θ 的一部分.在图中我们忽略了判定约束条件的步骤.

递归程序需要大的堆栈存贮被生成的断言序列,这使得许多问题虽然可以写出它们的递归描述,但计算它们实际上是不可能的.迭代程序计算时生成的断言序列是一个固定的常量,不需要大堆栈.图 2 显示了由第 2 节中的迭代程序(F1)—(F4)和起始断言$\rightarrow fact(2)$确

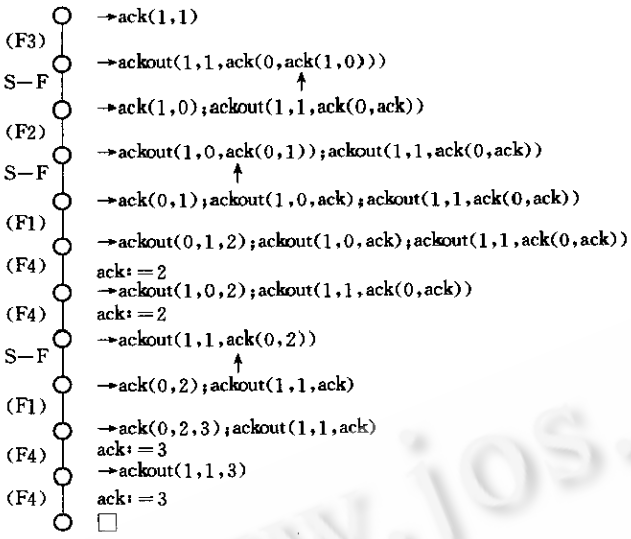

图1 用递归程序对ackermann's函数的计算

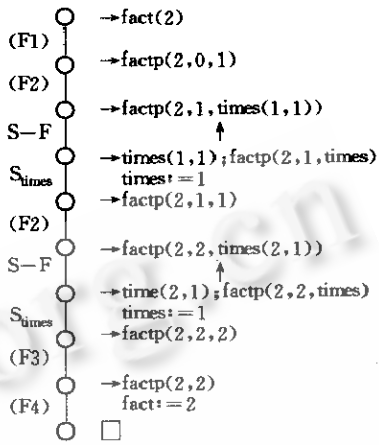

图2 阶乘函数的迭代计算

定的成功计算.弧上的标示意义除 S_{times} 外,其他与图 1 相同.S_{times} 表示由计算函数 *times* 的子句集确定的子计算并且计算结果以函数名 *times* 作为变量返回.

用逻辑解释,计算是带有函数调用的归结推导.一个计算的末尾断言是原始程序 S 和起始断言的逻辑结论.特别地,如果一个计算是成功的,那么它是 S 的一个反驳,即 S 的不可满足性的一个演示.任何传统的定理证明都不能被解释为上述意义下的计算.在上述意义下,一个推理系统是正确和完备的,指一个状态逻辑程序 S 和起始断言构成的子句集是不可满足的当且仅当推理系统有一个反驳.正确性和完备性的进一步讨论和证明需要长的篇幅,因此被省略.

最后,正如前面所看到的,证明过程由旧的断言或断言序列推导新的断言或断言序列是由底向上的证明过程.它不同于由旧的目标推导新目标的由顶向下的证明过程[10].

7 确定性

状态逻辑语言本质上是一种确定的程序设计语言.确定性由下面事实引起,即给定一个程序 S 和一个起始断言,仅有一个成功计算.下面考虑计算一个整数表的 *reverse* 函数的迭代程序

(F1) $rev(x)\rightarrow revp(x,x,NIL)$

(F2) $revp(x,y,z),\neg(y=NIL)\rightarrow revp(x,tail(y),cons(head(y),z))$

(F3) $revp(x,y,z),(y=NIL)\rightarrow revout(x,z)$

(F4) $revout(x,rev)\rightarrow$

其中 NIL 表示空表;$head(y)$ 表示表 y 的第一个元素;$tail(y)$ 表示去掉表 y 的第一个元素余下的表;$cons(x,y)$ 表示元素 x 加到表 y 的头部构成的表;$rev(x)$ 表示 x 是表;$revp(x,y,z)$ 表示表 x 和 y 的差表的反序表是 z;$revout(x,z)$ 表示表 x 的反序表是 z.图 3 显示了该

程序和起始断言→*rev*([1 2])确定的全部计算空间.

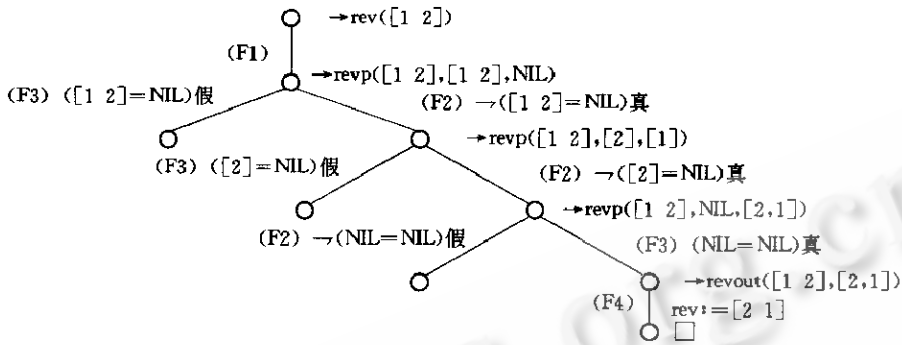

图3 计算reverse函数的全部计算空间

这个计算空间包含 3 个被截断的计算，仅有一个计算是成功的. 一个函数或过程调用可能有多个名字与其匹配的过程，然而在测试条件控制下仅有一个过程体被调用. 这是第一种确定性. 如图 3 所示，当一个测试为假时，一个计算被截断，使计算仅沿测试条件为真的弧构成的唯一路径进行，直到终止. 这使得成功的计算唯一地确定了所有过程的过程体的执行顺序而与过程，即子句之间的书写顺序无关. 如第 5 节中指出的，一个过程体或约束文字中每个项的所有子项之间的执行顺序由给定的函数调用规则唯一地确定. 这是第二种确定性. 状态逻辑的确定性使程序执行没有回溯.

参考文献

1 John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. Comm. of the ACM, 1978, **21(8)**: 613—641.

2 Guo Yike, Lock H C R. A classification scheme for declarative programming languages—syntax, semantics, and operational models. GMD—Studien N8. 182, August 1990.

3 John Darlington, Guo Yike, Helen Pull. A design space for integrating declarative languages. In: Darlington J, Dietrich R ed. Declarative Programming 1991, Sasbachwalden, 1992.

4 John Darlington, Guo Yike, Helen Pull. Introducing constraint functional logic programming. In: Darlington J, Dietrich R ed. Declarative Programming 1991, Sasbachwalden, 1992.

5 Kowalski R A. Predicate logic as programming language. Proc. IFIP Cong. 1974, North—Holland Pub. Co. Amsterdam, 1974: 568—574.

6 Morris J H. Real programming in function languages. Function Programming and Its Applications, Darlington, 1982.

7 Chang Chinliang, Lee Richard Chartung. Symbolic logic and mechanical theorem proving. National Institutes of Health, Bethesda, Maryland, 1973.

8 Manna Z, Mathematical theory of computation. McGRAW—HILL Book Company, 1974.

9 Lloyd J W. Foundation of logic programming. Germany, 1984.

10 Kowalski R A. Logic for problem solving. Memo No. 4, Department of Artificial Intelligence, University of Edinburgh, 1974.

STATE LOGIC PROGRAMMING LANGUAGE

Yan Zhixin

(*Department of Computer Science and Engineering, Beijing University of Aeronautics and Astronautics, Beijing* 100083)

Abstract A new pure logical language in clausal form was presented. The syntax, informal semantics, procedural interpretation of clauses and inference system based on constraint resolution were given. In this language, programs contain three types of variables: input variables, output variables and program variables to control resources of the computer; function symbols defined by programs can constitute terms or subterms and can be used as predicate symbols; the most general unification which is inefficient is not needed. According to these factors, programs implicate sequential, branched, iterative and recursive controls and it is easy to constitute efficient theorem—proving systems. It is believed that this language is a efficient, useful and practical, high—level, deterministic programming language with sound theoretical foundation.

Key words State logic language, call rule, splitting function call, constraint resolution, determinism.